Profiling & Parallelization

Lecture 21

Dr. Colin Rundel

Profiling & Benchmarking

profvis - lm() demo

```
n = 1e6
d = tibble(
  x1 = rt(n, df = 3),
  x2 = rt(n, df = 3),
  x3 = rt(n, df = 3),
  x4 = rt(n, df = 3),
  x5 = rt(n, df = 3),
) |>
  mutate(y = -2*x1 - 1*x2 + 0*x3 + 1*x4 + 2*x5 + rnorm(n))
```

```
profvis::profvis({
  lm(y~., data=d)
})
```

profvis - vector demo

```
profvis::profvis({
  data = data.frame(value = runif(5e4))

  data$sum[1] = data$value[1]
  for (i in seq(2, nrow(data))) {
    data$sum[i] = data$sum[i-1] + data$value[i]
  }
})
```

```
profvis::profvis({
  x = runif(5e4)
  sum = x[1]
  for (i in seq(2, length(x))) {
    sum[i] = sum[i-1] + x[i]
  }
})
```

Benchmarking - bench

```
 1 d = tibble(
 2   x = runif(10000),
 3   y = runif(10000)
 4 )
 5
 6 (b = bench::mark(
 7   d[d$x > 0.5, ],
 8   d[which(d$x > 0.5), ],
 9   subset(d, x > 0.5),
10   filter(d, x > 0.5)
11 ))
```

```
# A tibble: 4 × 6
  expression                   min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>              <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 d[d$x > 0.5, ]           74.5µs   89.5µs    11091.  236.52KB     36.0
2 d[which(d$x > 0.5), ]      76µs   90.8µs    10827.  271.91KB     65.4
3 subset(d, x > 0.5)       92.6µs  114.1µs     8756.  289.07KB     53.9
4 filter(d, x > 0.5)      249.5µs  292.2µs     3438.    1.48MB     21.3
```

Larger n

```r
d = tibble(
  x = runif(1e6),
  y = runif(1e6)
)

(b = bench::mark(
  d[d$x > 0.5, ],
  d[which(d$x > 0.5), ],
  subset(d, x > 0.5),
  filter(d, x > 0.5)
))
```

```
# A tibble: 4 × 6
  expression                   min   median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>              <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
1 d[d$x > 0.5, ]           7.21ms   7.52ms     133.     13.3MB     79.8
2 d[which(d$x > 0.5), ]    8.64ms   8.97ms     111.     24.8MB    144.
3 subset(d, x > 0.5)      11.16ms  11.55ms      86.0    24.8MB    108.
4 filter(d, x > 0.5)       8.38ms    9.5ms     103.     24.8MB     70.2
```

bench - relative results

```
1 summary(b, relative=TRUE)
```

```
# A tibble: 4 × 6
  expression              min median `itr/sec` mem_alloc `gc/sec`
  <bch:expr>            <dbl>  <dbl>     <dbl>     <dbl>    <dbl>
1 d[d$x > 0.5, ]         1      1         1.55      1        1.14
2 d[which(d$x > 0.5), ]  1.20   1.19      1.29      1.86     2.05
3 subset(d, x > 0.5)     1.55   1.54      1         1.86     1.53
4 filter(d, x > 0.5)     1.16   1.26      1.20      1.86     1
```

Example - BLAS and LAPACK

Statistics and Linear Algebra

An awful lot of statistics is at its core linear algebra.

For example:

- Linear regession models, find

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

- Principle component analysis
  - Find $T = XW$ where W is a matrix whose columns are the eigenvectors of $X^T X$.
  - Often solved via SVD - Let $X = U\Sigma W^T$ then $T = U\Sigma$.

Numerical Linear Algebra

Not unique to Statistics, these are the type of problems that come up across all areas of numerical computing.

- Numerical linear algebra ≠ mathematical linear algebra
- Efficiency and stability of numerical algorithms matter
  - Designing and implementing these algorithms is hard
- Don’t reinvent the wheel - common core linear algebra tools (well defined API)

BLAS and LAPACK

Low level algorithms for common linear algebra operations

BLAS

- **B**asic **L**inear **A**lgebra **S**ubprograms
- Copying, scaling, multiplying vectors and matrices
- Origins go back to 1979, written in Fortran

LAPACK

- **L**inear **A**lgebra **Pack**age
- Higher level functionality building on BLAS.
- Linear solvers, eigenvalues, and matrix decompositions
- Origins go back to 1992, mostly Fortran (expanded on LINPACK, EISPACK)

Modern variants?

Most default BLAS and LAPACK implementations (like R's defaults) are somewhat dated

- Written in Fortran and designed for a single cpu core
- Certain (potentially non-optimal) hard coded defaults (e.g. block size).

Multithreaded alternatives:

- ATLAS - Automatically Tuned Linear Algebra Software
- OpenBLAS - fork of GotoBLAS from TACC at UTexas
- Intel MKL - Math Kernel Library, part of Intel's commercial compiler tools
- cuBLAS / Magma - GPU libraries from Nvidia and UTK respectively
- Accelerate / vecLib - Apple's framework for GPU and multicore computing

R & BLAS / LAPACK

```
1 sessionInfo()
```

```
R version 4.5.2 (2025-10-31)
Platform: aarch64-apple-darwin20
Running under: macOS Tahoe 26.1

Matrix products: default
BLAS:   /System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.f
LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dy

locale:
[1] C.UTF-8/en_US.UTF-8/C.UTF-8/C/C.UTF-8/C.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] parallel  stats     graphics  grDevices utils     datasets
[7] methods   base

other attached packages:
 [1] lubridate_1.9.4 forcats_1.0.1   stringr_1.5.2   dplyr_1.1.4
```

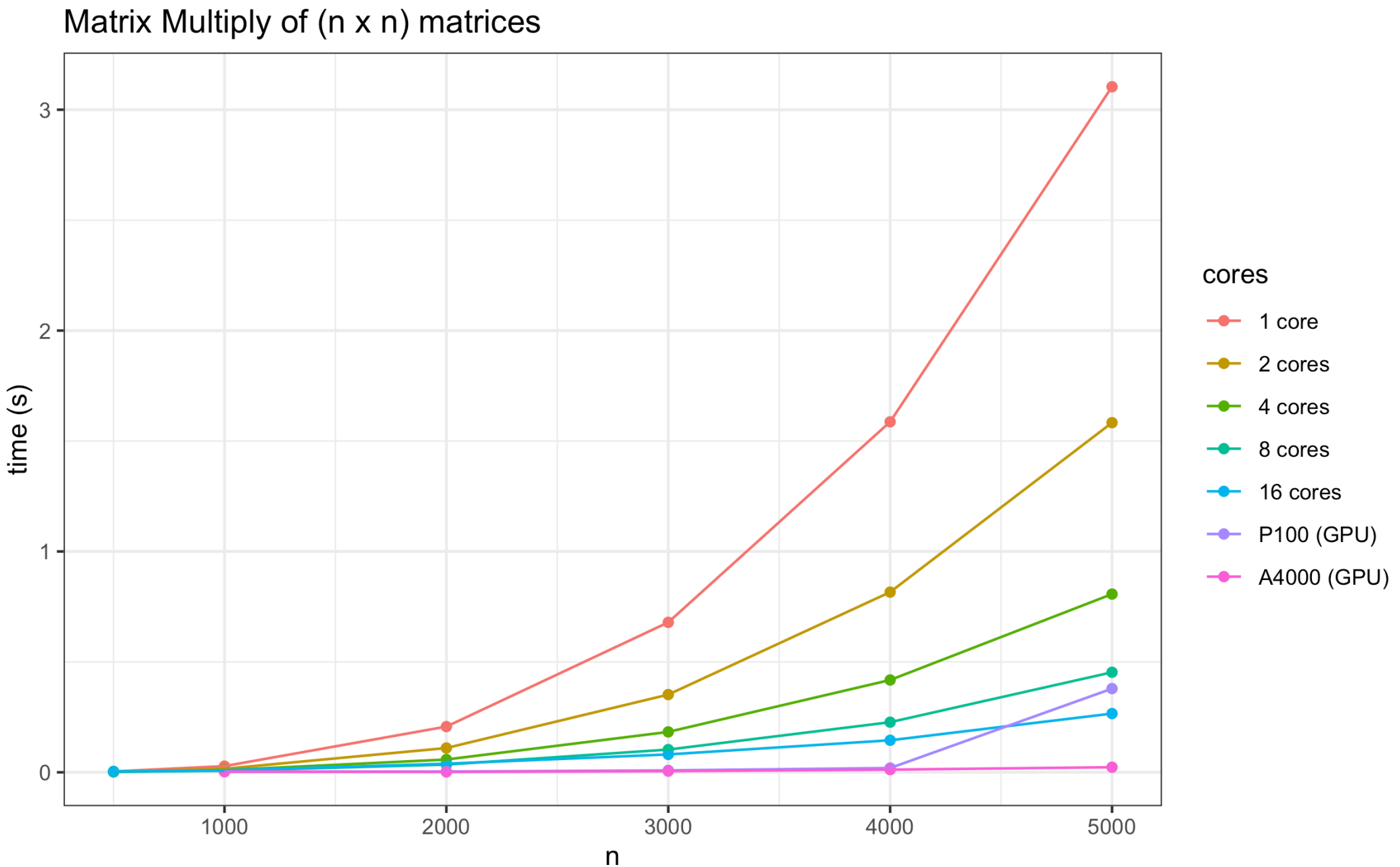
OpenBLAS Matrix Multiply Performance

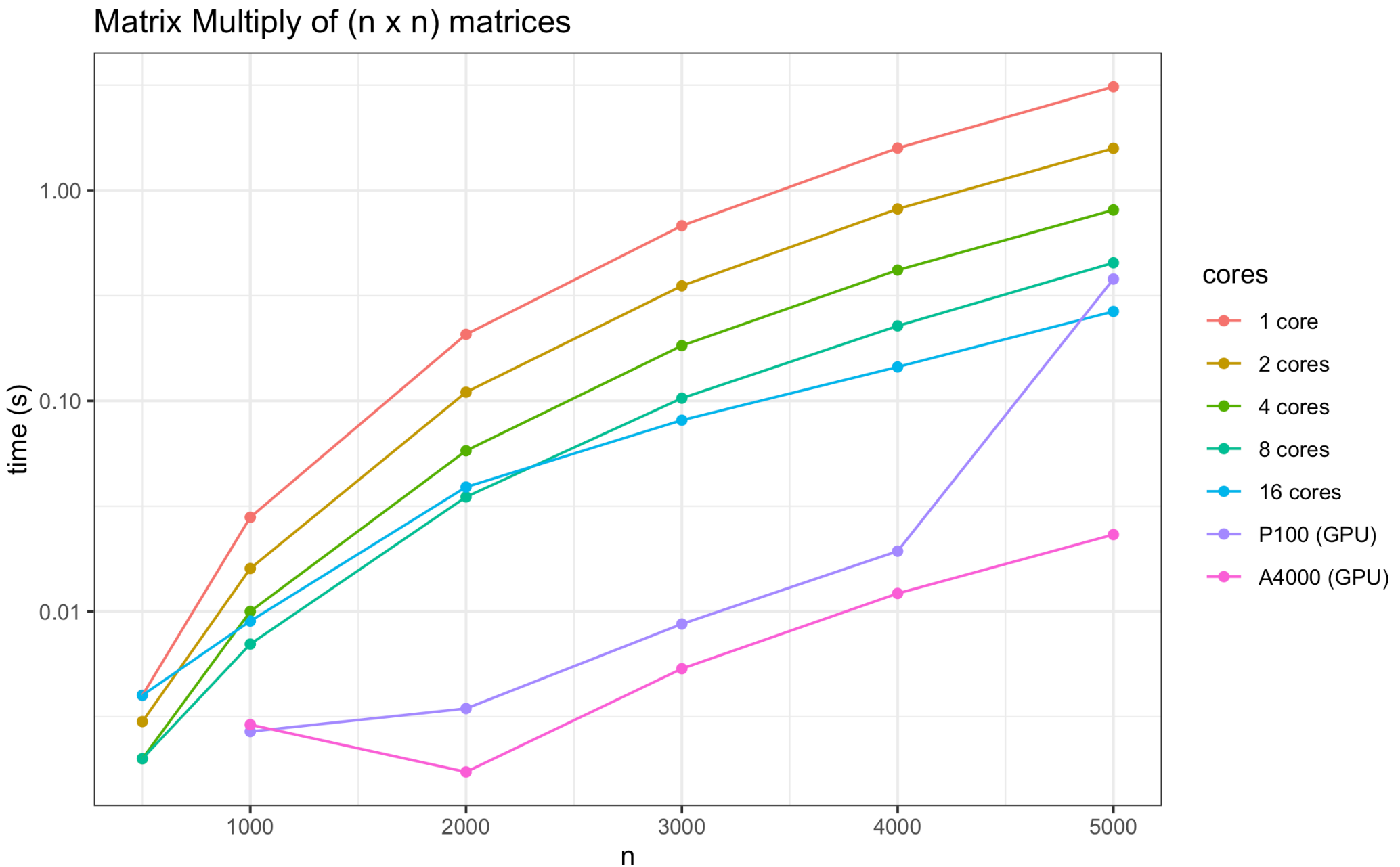
```
x = matrix(runif(5000^2),ncol=5000)

sizes = c(100,500,1000,2000,3000,4000,5000)
cores = c(1,2,4,8,16)

sapply(
  cores,
  function(n_cores) {
    flexiblas::flexiblas_set_num_threads(n_cores)
    sapply(
      sizes,
      function(s) {
        y = x[1:s,1:s]
        system.time(y %*% y)[3]
      }
    )
  }
)
```

n	1 core	2 cores	4 cores	8 cores	16 cores
100	0.000	0.000	0.000	0.000	0.000
500	0.004	0.003	0.002	0.002	0.004
1000	0.028	0.016	0.010	0.007	0.009
2000	0.207	0.110	0.058	0.035	0.039
3000	0.679	0.352	0.183	0.103	0.081
4000	1.587	0.816	0.418	0.227	0.145
5000	3.104	1.583	0.807	0.453	0.266


Matrix Multiply of (n x n) matrices
time (s)
0
1
2
3
1000
2000
3000
4000
5000
n
cores
1 core
2 cores
4 cores
8 cores
16 cores
P100 (GPU)
A4000 (GPU)


Matrix Multiply of (n x n) matrices
time (s)
1.00
0.10
0.01
1000
2000
3000
4000
5000
n
cores
1 core
2 cores
4 cores
8 cores
16 cores
P100 (GPU)
A4000 (GPU)

Parallelization in R

`parallel`

Part of the base packages in R

- tools for the forking of R processes (some functions do not work on Windows)
- Some core functions:
  - `detectCores`
  - `pvec`
  - `mclapply`
  - `mcparallel` & `mccollect`
- There are many other functions for creating and managing "clusters" of R processes, but we won't cover those here.

detectCores

Surprisingly, this detects the number of cores of the current system.

```
1 detectCores()
```

```
[1] 14
```

Take the reported number with a grain of salt / it is important to know somethimg about the hardware you are using. Modern CPUs are complicated and have many features (e.g. hyperthreading, P vs E cores, etc) that can make this number only part of the whole story.

This number also is not aware of other users / processes on the system / job scheduling constraints - so be considerate & conservative when choosing how many cores to use for your parallel computations.

pvec

Parallelize a vectorized function call, `FUN` must be a vectorized *pure* function (i.e. no side effects) that returns a vector the same length as `x`.

```
1 system.time(pvec(1:1e7, sqrt, mc.cores = 1))
```

```
   user  system elapsed 
  0.012   0.010   0.022 
```

```
1 system.time(pvec(1:1e7, sqrt, mc.cores = 4))
```

```
   user  system elapsed 
  0.228   0.748   0.899 
```

```
1 system.time(pvec(1:1e7, sqrt, mc.cores = 8))
```

```
   user  system elapsed 
  0.085   0.716   0.690 
```

```
1 system.time(sqrt(1:1e7))
```

```
   user  system elapsed 
  0.012   0.007   0.019 
```

pvec - bench::system_time

If you need more precise timing information, `bench::system_time` is a good alternative.

```
1 bench::system_time(Sys.sleep(.5))
```

```
process    real
   36µs   497ms
```

```
1 system.time(Sys.sleep(.5))
```

```
   user  system elapsed
  0.000   0.001   0.504
```

```
1 bench::system_time(pvec(1:1e7, sqrt, mc.cores = 1))
```

```
process    real
 25.9ms  25.6ms
```

```
1 bench::system_time(pvec(1:1e7, sqrt, mc.cores = 4))
```

```
process    real
  425ms   701ms
```

```
1 bench::system_time(pvec(1:1e7, sqrt, mc.cores = 8))
```

```
process    real
  623ms   899ms
```

```
1 bench::system_time(sqrt(1:1e7))
```

```
process    real
 23.8ms  23.4ms
```

Cores by size

```
cores = c(1,4,6,8,10)
order = 6:8
f = function(x,y) {
  system.time(
    pvec(1:(10^y), sqrt, mc.cores = x)
  )[3]
}

res = map(
  cores,
  function(x) {
     map_dbl(order, f, x = x) |>
      setNames(paste0("10^",order))
  }
) |>
  bind_rows() |>
  mutate(cores = cores) |>
  relocate(cores)
```

```
res
```

```
# A tibble: 5 × 4
  cores  `10^6` `10^7` `10^8`
  <dbl>   <dbl>  <dbl>  <dbl>
1     1 0.00100 0.0170  0.361
2     4 0.0870  0.708   7.81
3     6 0.093   0.699   6.90
4     8 0.0960  0.714   7.14
5    10 0.110   0.764   7.34
```

mclapply

Implements a parallelized version of `lapply`. It relies on forking and hence is not available on Windows unless `mc.cores = 1`.

```
1 system.time(rnorm(1e7))
```

```
   user  system elapsed 
  0.145   0.003   0.149 
```

```
1 system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 2)))
```

```
   user  system elapsed 
  0.190   0.693   0.770 
```

```
1 system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 4)))
```

```
   user  system elapsed 
  0.198   0.693   0.711 
```

```
1 system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 8)))
```

```
   user  system elapsed 
  0.207   0.754   0.727 
```

```
1 system.time(unlist(mclapply(1:10, function(x) rnorm(1e6), mc.cores = 10)))
```

```
   user  system elapsed 
  0.212   0.763   0.732 
```

mcparallel

Asynchronously evaluation of an R expression in a separate process

```
1 m = mcparallel(rnorm(1e6))
2 n = mcparallel(rbeta(1e6,1,1))
3 o = mcparallel(rgamma(1e6,1,1))
```

```
1 str(m)
```

```
List of 2
 $ pid: int 90613
 $ fd : int [1:2] 5 8
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

```
1 str(n)
```

```
List of 2
 $ pid: int 90614
 $ fd : int [1:2] 6 10
 - attr(*, "class")= chr [1:3] "parallelJob" "childProcess" "process"
```

mccollect

Checks `mcparallel` objects for completion - if finished the values are returned, otherwise the calling R process is blocked until they are done.

```
1 str(mccollect(list(m,n,o)))
```

```
List of 3
 $ 90613: num [1:1000000] -0.4154 0.0169 2.0027 -0.8437 -0.308 ...
 $ 90614: num [1:1000000] 0.346 0.692 0.128 0.36 0.674 ...
 $ 90615: num [1:1000000] 1.715 0.153 0.444 0.778 0.746 ...
```

mccollect - waiting

If you don't want to block the calling R process, you can use the `wait = FALSE` argument. This will return `NULL` if the process is not yet complete.

```
1 f = function(n) {
2   Sys.sleep(5)
3   mean(rnorm(n))
4 }
5 p = mcparallel(f(1e5))
```

```
1 mccollect(p, wait = FALSE)
```

```
NULL
```

```
1 mccollect(p, wait = FALSE, timeout = 5)
```

```
$`90616`
[1] 0.0006274726
```

```
1 mccollect(p, wait = FALSE)
```

```
Warning in selectChildren(jobs, timeout): cannot wait for child 90616
as it does not exist
```

```
NULL
```

Aside - RNG and Parallelization

Random number generation in parallel contexts requires special consideration:

- Multiple processes using the same RNG seed produce identical "random" sequences
- Introducing correlation into our RNG invalidates statistical results (e.g., bootstrap, Monte Carlo)
- For most `parallel` package functions, the default behavior ensures proper RNG handling, but be aware of this issue when writing custom parallel code.
- "Reproducibility" can be a function of both your data and core / worker size.

Example

```
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)), mc.cores = 3
) |>
  unlist()
```

```
[1] 0.9617990 0.9984449 1.0023891 0.9206979
```

```
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)), mc.cores = 3
) |>
  unlist()
```

```
[1] 1.0630479 1.0276156 1.0110946 0.9407919
```

```
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)),
  mc.cores = 3, mc.set.seed = FALSE
) |>
  unlist()
```

```
[1] 1.004405 1.004405 1.004405 1.032187
```

```
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)),
  mc.cores = 3, mc.set.seed = FALSE
) |>
  unlist()
```

```
[1] 1.004405 1.004405 1.004405 1.032187
```

```
RNGkind("L'Ecuyer-CMRG")
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)), mc.cores = 3
) |>
  unlist()
```

```
[1] 0.9350555 1.1025750 1.0046105 0.9288966
```

```
RNGkind("L'Ecuyer-CMRG")
set.seed(1234)
mclapply(
  1:4, function(x) sd(rnorm(100)), mc.cores = 3
) |>
  unlist()
```

```
[1] 0.9350555 1.1025750 1.0046105 0.9288966
```

parallel alternatives & predecessors

Historical landscape

Early packages

- `snow` (2003) - Simple Network of Workstations, cluster-based parallelization
- `multicore` (2009-2014) - Fork-based parallelization for Unix-like systems
- `foreach` (2009) - Iterative parallel execution with pluggable backends

Consolidation

- `parallel` - Merged `snow` and `multicore` into base R (R 2.14.0)

Modern alternatives

- `foreach` + backends (`doParallel`, `doMC`, `doFuture`) - Still widely used w/ flexible backends
- `future` (2015) - Modern async framework, support for local/remote/cluster computing
- `mirai` (2022) - Lightweight async evaluation, powers modern tidyverse parallelization

mirai

is a modern framework for asynchronous parallel computing in R

- Built on nanonext & NNG messaging (IPC, TCP, TLS)
- Scales to millions of tasks with minimal overhead
- Transparent evaluation with error handling
- Works locally, remote (SSH), and on HPC clusters

Core functions:

- `mirai()` - evaluate R expressions asynchronously
- `daemons()` - persistent background processes
- `mirai_map()` - parallel map operations

Basic `mirai()` usage

Just like `mcparallel`, we can eval R expressions without blocking the session,

```
library(mirai)
m = mirai(
  {
    Sys.sleep(time)
    rnorm(5L, mean)
  },
  time = 2L,
  mean = 4.5
)
```

```
m
```

```
< mirai [] >
```

```
m$data
```

```
'unresolved' logi NA
```

```
unresolved(m)
```

```
[1] TRUE
```

```
m[]
```

```
[1] 6.121372 4.118744 5.614244 3.363124 3.
```

```
m$data
```

```
[1] 6.121372 4.118744 5.614244 3.363124 3.
```

```
m
```

```
< mirai [$data] >
```

mirai objects

> A mirai is either unresolved if the result has yet to be received, or resolved if it has. unresolved() is a helper function to check the state of a mirai.
>
> The result of a mirai m is available at `m$data` once it has resolved. Normally this will be the return value of the evaluated expression. If the expression errored, crashed, or timed out then this will be an 'errorValue' instead. See the section Error Handling below.
>
> Rather than repeatedly checking unresolved(m), it is more efficient to wait for and collect its value by using m[].
>
> ...
>
> If execution in a mirai fails, the error message is returned as a character string of class 'miraiError' and 'errorValue' to facilitate debugging.
>
> `is_mirai_error()` may be used to test for mirai execution errors.

Error handling

Failed evaluations return `miraiError` objects,

```
m = mirai({
  stop("Something went wrong!")
})
```

```
m[]
```

```
'miraiError' chr Error: Something went wrong!
```

```
is_mirai_error(m$data)
```

```
[1] TRUE
```

```
m$data$message
```

```
[1] "Something went wrong!"
```

```
m$data$stack.trace
```

```
[[1]]
stop("Something went wrong!")
```

```
m$data$condition.class
```

```
[1] "simpleError" "error"       "condition"
```

Notes on evaluation

The expression passed to `mirai()` will be evaluated in *a separate R process in a clean environment (not the global environment)*. Any variables, functions or other objects that are needed must be supplied via `...` and/or `.args`.

- Use `...` for simple arguments that are substituted into the experssion before interpretation
- Use `.args` for functions, large objects, etc.

Similarly, if you are using functions from non-base packages you must use namespaced calls (e.g. `dplyr::select()`), or else the package should be loaded explicitly as part of `.expr`.

Note that when using daemones the function `everywhere()` can be used to evaluate an expression on all connected daemons, this can be useful to load packages or set options.

daemons()

We can create persistent background processes to handle our `mirai()` requests, this reduces startup overhead for each task.

```
daemons(0)
a = mirai({
  Sys.sleep(3)
})
b = mirai({
  Sys.sleep(2)
})
c = mirai({
  Sys.sleep(1)
})

system.time(
  call_mirai(list(a, b,
)
```

```
   user  system elapsed
  0.001   0.001   3.016
```

```
daemons(1)
a = mirai({
  Sys.sleep(3)
})
b = mirai({
  Sys.sleep(2)
})
c = mirai({
  Sys.sleep(1)
})

system.time(
  call_mirai(list(a, b,
)
```

```
   user  system elapsed
  0.000   0.000   5.939
```

```
daemons(3)
a = mirai({
  Sys.sleep(3)
})
b = mirai({
  Sys.sleep(2)
})
c = mirai({
  Sys.sleep(1)
})

system.time(
  call_mirai(list(a, b,
)
```

```
   user  system elapsed
   0.00    0.00    2.93
```

`call_mirai()` vs `race_mirai()`

```
daemons(4)
```

Call waits for all tasks to complete

```
a = mirai({
  Sys.sleep(3)
  "A"
})
b = mirai({
  Sys.sleep(2)
  "B"
})
c = mirai({
  Sys.sleep(1)
  "C"
})

call_mirai(list(a, b, c))
c(a$data, b$data, c$data)
```

```
[1] "A" "B" "C"
```

Race waits for the first task to complete

```
a = mirai({
  Sys.sleep(3)
  "A"
})
b = mirai({
  Sys.sleep(2)
  "B"
})
c = mirai({
  Sys.sleep(1)
  "C"
})

race_mirai(list(a, b, c))
c(a$data, b$data, c$data)
```

```
[1] NA  NA  "C"
```

`mirai_map()`

This function is similar to `purrr::map()`, but instead of a list of values, returns a 'mirai_map' object, which is a list of mirai.

A `mirai_map()` call returns almost immediately. The results of a mirai_map x may be collected using `x[]` or `collect_mirai(x)`, the same as for a mirai. This waits for all asynchronous operations to complete if still in progress.

Key advantages

1. Returns immediately with all evaluations taking place asynchronously. Printing a 'mirai map' object shows the current completion progress.
2. The '.promise' argument allows a promise action to be registered against each iteration. This can be used to perform side-effects when each iteration completes (such as checkpointing or recording a progress update).
3. Returns evaluation errors as 'miraiError' or 'errorValue' as the case may be, rather than causing the entire map to fail. This allows more efficient recovery from partial failure.
4. Does not rely on a 'chunking' algorithm that attempts to split work into batches according to the number of available daemons, as implemented for instance in the parallel package. Chunking cannot take into account varying or unpredictable compute times over the indices, which mirai scheduling is designed to deal with optimally.

Scheduling Examples

```
daemons(4)
waits = c(1, 1, 4, 4, 1, 1, 1, 4)
```

```
system.time(
  mirai_map(waits, Sys.sleep)[]
)
```

```
   user  system elapsed
  0.001   0.001   6.017
```

```
system.time(
  mclapply(waits, Sys.sleep, mc.cores = 4)
)
```

```
   user  system elapsed
  0.002   0.033   8.029
```

```
daemons(4)
n = sample(1:8)
f = function(n) mean(rnorm(10^n))
```

```
system.time(
  mirai_map(n, f)[]
)
```

```
   user  system elapsed
  0.000   0.001   2.562
```

```
system.time(
  mclapply(n, f, mc.cores = 2)
)
```

```
   user  system elapsed
  0.028   0.028   3.019
```

purrr's `in_parallel()` (experimental)

Enables parallel computation in purrr map functions using mirai,

```
1 system.time({
2   x = mclapply(
3     1:10,
4     \(x) rnorm(1e6),
5     mc.cores = 5
6   ) |>
7     unlist()
8 })
```

```
   user  system elapsed 
  0.455   0.775   0.937 
```

```
1 mean(x)
```

```
[1] 7.349909e-05
```

```
1 sd(x)
```

```
[1] 1.000287
```

```
1 system.time({
2   daemons(5)
3   x = map(
4     1:10,
5     in_parallel(\(x) rnorm(1
6   ) |>
7     unlist()
8 })
```

```
   user  system elapsed 
  0.118   0.151   0.576 
```

```
1 mean(x)
```

```
[1] 0.0003887193
```

```
1 sd(x)
```

```
[1] 1.000248
```

Example - Bootstraping

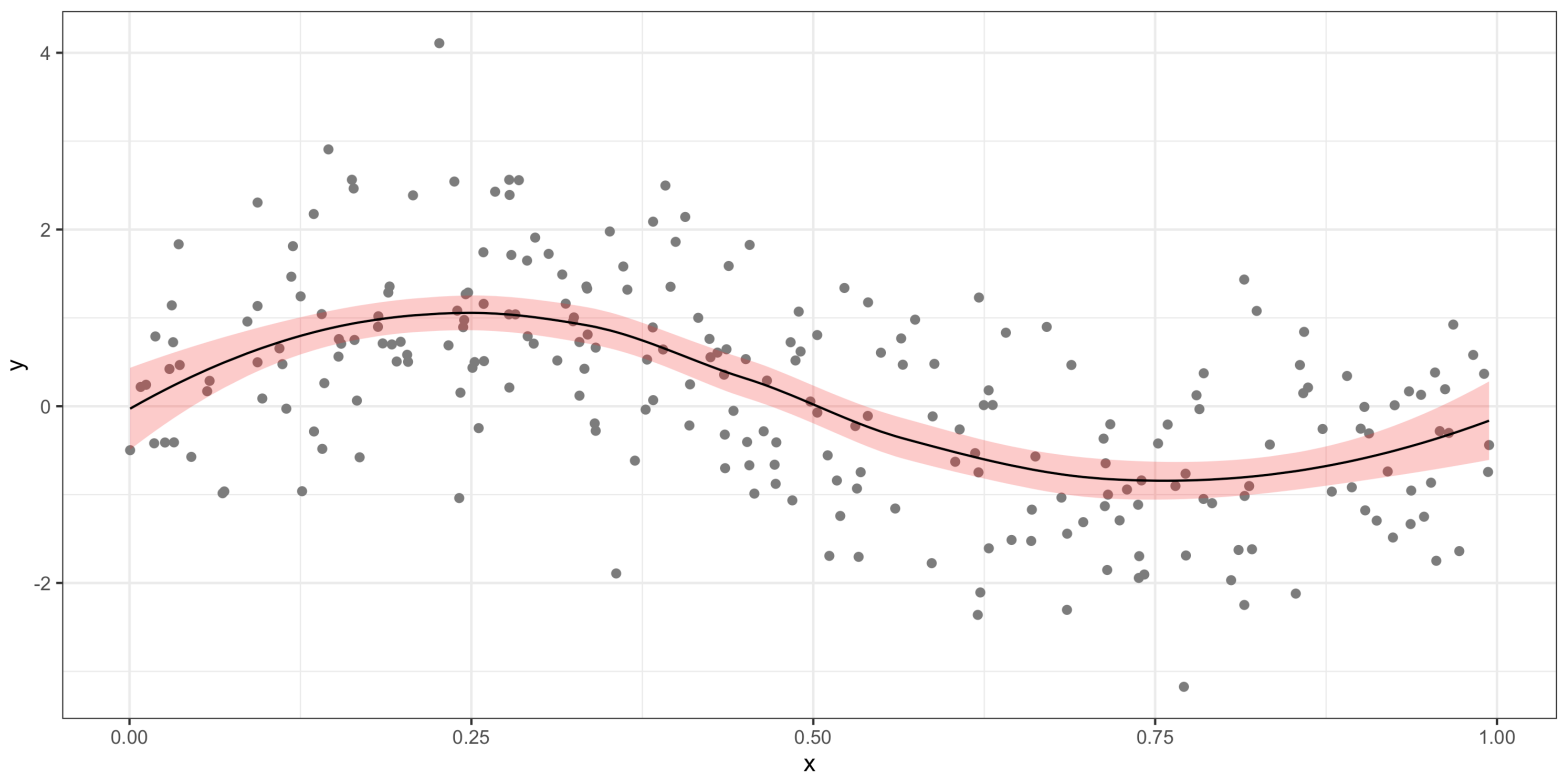
Bootstrapping is a resampling scheme where the original data is repeatedly reconstructed by taking a samples of size *n* (with replacement) from the original data, and using that to repeat an analysis procedure of interest. Below is an example of fitting a local regression (`loess`) to some synthetic data, we will construct a bootstrap prediction interval for this model.

```r
set.seed(3212016)
d = data.frame(x = runif(250)) |>
  mutate(y = sin(2*pi*x) + rnorm(length(x)))

l = loess(y ~ x, data=d)
p = predict(l, se=TRUE)

d = d |> mutate(
  pred_y = p$fit,
  pred_y_se = p$se.fit
)
```

```r
ggplot(d, aes(x,y)) +
  geom_point(color="gray50") +
  geom_ribbon(
    aes(ymin = pred_y - 1.96 * pred_y_se,
        ymax = pred_y + 1.96 * pred_y_se),
    fill="red", alpha=0.25
  ) +
  geom_line(aes(y=pred_y)) +
  theme_bw()
```



Bootstraping Demo

What to use when?

Optimal use of parallelization / multiple cores is hard, there isn't one best solution

- Don't underestimate the overhead cost
- Experimentation is key
- Measure it or it didn't happen
- Be aware of the trade off between developer time and run time